An Autotuning Protocol to Rapidly Build Autotuners

Guangming Tan, Junhong Liu, Yulong Luo
State Key Laboratory of Computer Architecture
Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
{tgm,liujunhong,luoyulong}@ncic.ac.cn

ABSTRACT

Performance tuning is becoming a challenging engineering work as architectural complexity grows. Autotuning is emerging as a critical technique to achieve high portable performance. However, the current approach to construct autotuners doesn't appeal to common users. In order to simplify the use of autotuning, we propose an autotuning protocol to achieve a better abstraction while developing a performance tuning and knowledge managing suite (PAK) through encapsulation. The protocol specifies five procedures extractor, producer, optimizer, evaluator and learner to assemble an autotuner rapidly. The autotuning programming interfaces for the protocol are implemented as a fundamental infrastructure to achieve modularity and reusability. The infrastructure provides customizable mechanism and deploys performance knowledge database for seamlessly leveraging performance knowledge mining. In the case studies of stencil and sparse matrix computations, PAK only needs to write tens of lines code to assemble the autotuners, and demonstrates higher productivity compared to the traditional approaches.

1. INTRODUCTION

The emerging multi/many-core technique brings more complexity and diversity of architecture and program model, as a consequence, it increases difficulty to develop high performance programs achieving reasonable efficiency, not speaking of extreme efficiency for either Exascale supercomputing or energy-constraint embedded computing. A conventional way to tune code by hand is trick-intensive and requires programmers to be highly knowledgeable about the relationship between software and its mapping to hardware. Although such a hand-tuned code may achieve extremely high performance, it is usually not performance portable across different execution context for the following reasons:

- *Input Variation:* Many applications have varying behaviors (i.e., locality, parallelism) based on their inputs. For example, a 7 point stencil application (without any optimization) exhibits different data locality behaviors when the size of input data changes from 256*256*512 to 512*512*1024, the performance difference has 10% percent X86 CPUs. The variant is caused by input, but correlated with many other factors of architecture and application, which are complicated for analyzing and optimizing alone.
- *Compiler Variation:* On currently computer system, compiler or programming model plays pivotal role on performance tuning. Not speaking of performance gap caused by different compilers (i.e., Intel C compiler and GNU GCC), a meticulous selection of one compiler's options may lead to several times of performance improvement. Considering that the huge number of modern compiler options, it is prohibitive to tune a optimal configuration by hand.
- *Hardware Variation:* With respect to the evolvement of hardware, optimal code on today's architecture is almost certain to be suboptimal on future's one. In fact, even on the different implementations of the same ISA, i.e., Intel and AMD X86 CPUs, their respective optimal codes may be significantly changes from each other. Obviously, it is extremely both time and human-resource consuming to hand-tune codes on every platform.

Recently, a critical technology for dealing with the significant changes is autotuning technique that acts as either an auto-tuning library or adaptive performance tuning framework for some specific application domain. However, it is not an easy work to implement an autotuning system or autotuner for a specific demand. We are facing with enormous challenges in devoloping and assembling an autotuner, even if they could make use of the existing works including tools and algorithms.

- *There is a lack of a modular, reusable and general approach to construct an autotuner.* To date, people have developed a series of autotuning libraries and frameworks, some of which (e.g., FFTW [2], ATLAS [15]) have been extensively used in accelerating various applications. However, the implementations used in one autotuning system cannot be directly reused in another one. In other words, we have to re-implement some autotuning strategies from scratch even if they have successfully applied in existing autotuners. To promote its popularization, several work pay their initial efforts on simplifying construction of autotuner.

OpenTuner [1] proposes a fully-customizable configuration representations, an extensible technique representation to allow for domain-specific techniques, and an easy to use interface for communicating with the program to be autotuned. This trend indicates that it is feasible to develop an autotuning methodology which is of modularity, reusability and generality.

- *There is a lack of a uniform, extensible, and customizable approach to preserve performance knowledge.* Performance optimization/tuning is an nontrivial work. Especially for the performance critical algorithms like stencil computation and sparse matrix operations, we have witnessed massive published papers on tuning their performance in every generations of processors in the past several decades. However, its effectiveness is often determined by prior knowledge of programmers. The knowledge here comprises performance data, optimization parameters, and characteristics of a specific program and machine. The most advantage of autotuning technique is to lower requirement of the knowledge for programmers. For example, the search-based autotuning [5] acquires knowledge of the best optimization strategies using empirical trials while the machine learning based one [7] stores the knowledge in a statistical model during the training phase. Unfortunately, the knowledge has different formats and organizations for each tool and algorithm so that it is difficult to facilitate a comprehensive optimization search in an autotuning system.

As a consequence, most of autotuning related components, such as static analysis tools [8], search algorithms [1], machine learning models [7], domain specific compilers [11, 3, 13, 10], highly-tuned algorithm libraries [8] and measure tools [9], cannot easily recognize each other, as they use different input and output interface. This fact hampers the extensive use of autotuning technique.

In this work, we propose a performance tuning and knowledge managing suit (**PAK**) to overcome these problems. This work targets to an infrastructure to construct modular autotuners. Our novelty is highlighted by an autotuning protocol of five abstraction modules–*extractor*, *producer*, *optimizer*, *evaluator* and *learner* to assemble an autotuner rapidly. We define their interfaces and specification of corresponding input/output for customizing modules with existing tools. With the five modules in mind, users can easily build an autotuning skeleton, and then instantiate it by either selecting some module implementation available in the infrastructure or providing their own implementation as a plugin. As we know, performance knowledge represents the inherent connection between a given instance (e.g., program, platform) and its optimization variants. Thus, a cornerstone of our methodology is performance knowledge database, which stores the knowledge produced during the course of autotuning. The database supports four basic data types, which expresses the knowledge in a uniform way. The organization of the database is extensible, and support the addition of new knowledge type. Moreover, it take uses of the input and output interface for new tools to automatically recognize the custom knowledge and store them to database. The main contributions of this paper are:

- We abstract a protocol of building an autotuner, which is composed of five basic procedures representing commonality of autotuning system. We further define autotuning programming interface (API) for constructing an autotuner with the protocol. They act as an infrastructure to support modularized autotuners.
- We propose a design of performance knowledge database, which provides a customize, extensible, and uniform way to automatically preserve knowledge generated in process of autotuning. It provides interface to describe performance data by defining customized tools.
- We demonstrate the use of PAK by building autotuners for stencil computation and sparse matrix-vector multiplication(SpMV), respectively. With PAK, users only need to write tens of lines code to assemble the autotuners. Compared to the corresponding traditional autotunes, PAK show higher productivity and comparable performance as well.

To the best of our knowledge, *it is the first time that autotuning methodology are generalized to composable procedures which can be modularized as an infrastructure leveraging performance knowledge database.* This paper is advocating an unified protocol of developing autotuners, instead of presenting a comprehensive implementation of infrastructure which relies on the effort of community.

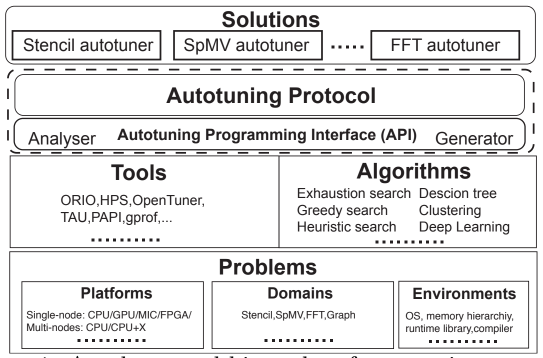


Figure 1: An abstracted hierarchy of autotuning systems.

2. AUTOTUNING PROTOCOL

In the current design of an autotuning system, each phase is specific to the targeted problem. In fact, an autotuning system can be hierarchically structured as shown in Figure 1. By dissecting the internal structure of most autotuning systems, we observe that they can be decomposed into several components.

Logically, the domain-specific autotuners can be formulated as *solutions* to some *problems*. Here, the problem is defined as: given an application *domain*, find its optimal implementation on any computing *platform* and execution *environment*. In order to help solve the problem, people have developed a series of *tools* and *algorithms*. Ideally, with these tools and algorithms available at hand people could easily build an autotuner. In fact, for a real-world implementation they only work case by case and still need heavy effort of reengineering. Therefore, we argue that it's necessary to design a protocol of constructing autotuners. With the protocol, an unified interface could be defined to programming any autotuner.

The idea of hierarchy inspires us to develop an infrastructure to modularly construct an autotuner and comprehensively preserve the autotuning knowledge. By integrating existing and emerging tools and algorithms, and employing a customizable, extensible and uniform knowledge database, it could greatly simplify the reusing of existing works and decrease the efforts of building an autotuner. Given a specific problem, traditional methodology requires a series of tedious procedures, which include analyzing problem, selecting/creating tools, implementing algorithm, defining interface and so on. By contrast, with PAK methodology, developers only need pay attention to analyzing problem, selecting/customizing modules and assembling autotuners.

The fundamental idea of PAK is an autotuning protocol composed of five customizable modules which are abstracted from the most of autotuning systems. In the protocol, we provide a set of abstract interfaces for the five elemental components, which should be customized by users to construct an autotuner. These interfaces state the implicit workflow of an autotuner with PAK as shown in Figure 2. The core modules, *extractor*, *producer*, *optimizer*, *evaluator* and *learner*, cooperatively work as follows.

- *Extractor* contains one or more analyzers which are integrated in a loosely-coupled way. It applies them to characterize a given running instance from differnent levels of algorithm, architecture and input, and outputs its features as result.

- *Producer* is responsible to generate an optimization solution candidate in each tuning step. It has an input interface that receives the features, the tuning step and the tuning score, and has the ability to realize any autotuning search algorithms, including both search-based approach and machine learning-based one.

- *Optimizer* gets a parameter, and employs a set of generators to perform the corresponding optimization strategy.

- *Evaluator* takes the optimized running instance as input. It comprises of one or more dynamic analyzers to measure the optimization result. Each of the features in these dynamic analyzers is related to a specific score function, which appraises the optimization result according to the preset object. If the appraisement satisfies the object, the autotuning stops and outputs the optimization result. Otherwise, the appraisement result is sent to the *Producer* and a new tuning step begins.

- *Learner* obtains the tuning data from the performance knowledge database, and builds machine learning models which facilitate the parameter generation of the machine learning-based *Producer*s. The knowledge data, including those generated during the course of tuning, are persevered in a database using a uniform format.

A remarkable advantage of PAK is the extensibility that promises an easy employment of the third-party tools. These tools provide measurement, analysis and optimization capabilities for a running instance during a tuning step, and produce valuable data. We refer to them as *knowledge data* because they contain characterization of the running instance, parameters of optimization and measurement of optimized

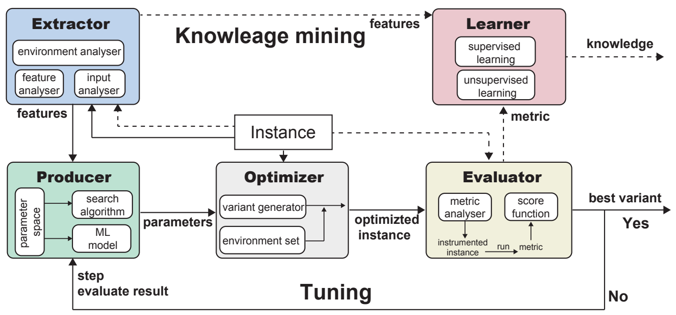


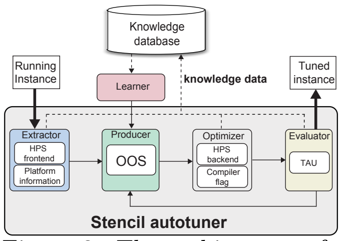
Figure 2: The workflow of an autotuner with PAK.

running instance. The knowledge data encodes a tuning step from the aspects of characterization of running instance, optimization strategy, and measurement result. However, most tools use fixed knowledge data based on their own formats, which are not recognized by each other.

PAK provides an interface for defining custom tools including a FDF describing knowledge data and a launch script standardizing the input and output files. The knowledge data are understood automatically by PAK and converted to the format that is compatible with the knowledge database. The knowledge database is implemented in a scalable organization form and employs an uniform format that consists of four data type to express the knowledge data. With these features, PAK implements a customizable, extensible and uniform knowledge database to preserve and access the knowledge data.

3. CASE STUDIES

In this section, we apply PAK to two performance critical problems– stencil computation and sparse matrix-vector multiplication (SpMV), which are extensively used in high performance computing applications. In order to demonstrate the high productivity of PAK, we re-implement the two autotuners by adopting the already developed tools and algorithms from their corresponding autotuning systems [7, 8]. Figure 3 and 4 show the architectures and implementations of the autotuners with PAK. We conduct experiments on a 16-core SMP system integrating two Intel Xeon E5-2670 multicore CPUs. The compiler is Intel compiler version 13.1.



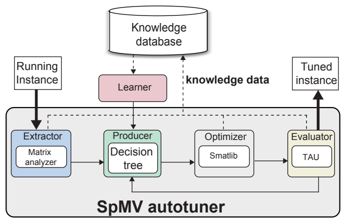
```
#customize an Extractor object'ext' and an Optimizerobject'opt'
ext=PAK.Extractor$new(list(hps_frontend=c(" FD" ," RD" ," AN" ," LR" ," AC"
   ," DT" ," PS" ," IT" ),platforminfo=c(" core" ," freq" ," cache" ," bd" )))
opt=PAK.Optimizer$new(generator.name=" hpsgen" )

#customize an Producer object'pro' and an Evaluator object'eva'
pro=PAK.Producer.Greedy$new(parameter.list)
eva=PAK.Evaluator$new(sub .evaluators=list (tau=
    list (P _WALL _CLOCK _TIME=function (x){ if(x> 0) return (-x)  else return (0)} )))

#assemble them togetherand performautotuning
for(app in training.stencils)
{
  tuning=PAK.Tuner$new(app=app,optimizer=opt,evaluator=eva,producer =pro)
  tuning$tune()
}
```

Figure 3: The architecture of stencil autotuner with PAK and its implementation.

The benchmark set consists of five stencil computation applications, which include FDTD [6], HEAT [3], WAVE [12], POISSON [14] and HIMENO [4]. Figure 5 plots the performance in Gflops (Giga floating-points per second) on CPU. The five baseline programs achieve 1.52 Gflops, 3.0Gflops, 1.59Gflops, 2.11Gflops and 1.84Gflops, respectively. As a comparison, our autotuner system achieves 10Gflops, 11.4Gflops,



```
#customize  an Extractor  object 'ext' and an Optimizer  object 'opt'
ext=PAK.Extractor   $new (list (spmv _extrator=  c (" M" ," N" ," NNZ" ," Ndiags" ,
    " NTdiags_ratio" ," ER_DIA" )))
opt=PAK.Optimizer    $new (generator.name=" smatlib" )
#customize  a Producer  object 'pro1' using  exhaustion  and an Evaluator  object 'eva'
pro1=PAK.Producer.Exhaustion     $new (list (method _name=" ' MV_COO' ,' MV_CSR'
      ,' MV_CSC' ,' MV_DIA' ;" ))
eva=PAK.Evaluator    $new (sub .evaluators=  list (tau=
     list (P _WALL  _CLOCK   _TIME=  function  (x) { if(x > 0)  return  (0 – x)  else return   (0) } )))
#perform  autotuning  for training
for (app in training.matrixs)
{
  tuning=PAK.Tuner   $new (app=app,optimizer=opt,evaluator=eva,producer =pro1,
           need.store=TRUE)
  tuning $tune()
}
#learn  the  model
pro2=PAK.Producer.DecisionTree      $new ()
pro2 $trainModel(trainingData," " ," method        _name" )
#autotuning  using  generated  decision  – tree  model
for (app in test.matrixs)
{
  t=objectives[[app]]
  eva2=PAK.Evaluator    $new (sub .evaluators=  list (tau= list (P _WALL  _CLOCK   _TIME=
         function  (x) { if(x > t)return  (t– x)  else return   (0) } )))
  tuning=PAK.Tuner   $new (app=app,extractor=ext,optimizer=opt,evaluator=eva,
           producer =pro2)
  tuning $tune()
}
```

Figure 4: The architecture of SpMV autotuner with PAK and its implementation.

3.7Gflops, 8Gflops and 7.5Gflops. The speedups are 6.6x for FDTD, 3.8 for HEAT, 2.3x for HIMENO, 3.8x for Poisson and 4.2x for WAVE. With multicore parallelism the speedups increase to 5.4x for FDTD, 6.2x for HEAT, 11.2x for HIMENO, 7.7x for Poisson and 5.4x for WAVE.

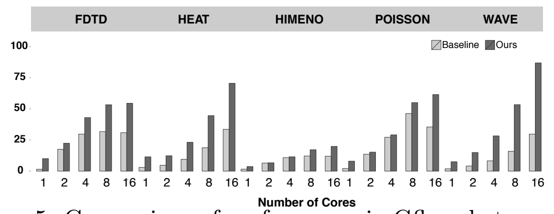


Figure 5: Comparison of performance in Gflops between the baseline and the autotuned implementation. The problem size is $256 \times 256 \times 512$.

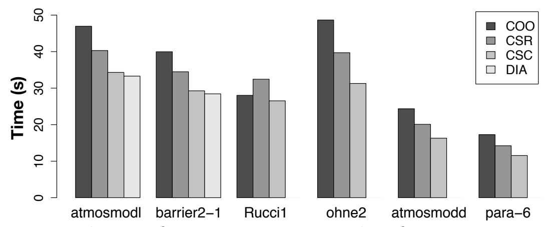


Figure 6: The performance in seconds of running time are predicted by the SpMV autotuner.

We list the performance of the six matrixes in test set in Figure 6. The x-axis presents the six matrixes, which are atmosmodl, barrier2-1, Rucci1, ohne2, atmosmodd and para-6, and the y-axis is their execution time. For the matrixes of atmosmodl and barrier2-1, as the implementation of DIA costs the least time, it is the best implementation. For the rest of six matrixes, the CSC implementation has the fastest speed.

4. REFERENCES

[1] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe. Opentuner: An extensible framework for program autotuning. In *International Conference on Parallel Architectures and Compilation Techniques*, Edmonton, Canada, August 2014.

[2] M. Frigo and S. G. Johnson. The design and implementation of fftw3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[3] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan. A stencil compiler for short-vector simd architectures. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 13–24. ACM, 2013.

[4] R. Himeno. Himeno benchmark, 2011.

[5] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[6] K. S. Kunz and R. J. Luebbers. *The finite difference time domain method for electromagnetics*. CRC press, 1993.

[7] J. Li, G. Tan, M. Chen, and N. Sun. Smat: an input adaptive auto-tuner for sparse matrix-vector multiplication. In *ACM SIGPLAN Notices*, volume 48, pages 117–126. ACM, 2013.

[8] Y. Luo, G. Tan, and N. Sun. Fast: An fast stencil autotuning framework based on optimal space model. In *International Conference on Supercomputing*, 2015.

[9] A. D. Malony, J. Cuny, and S. Shende. Tau: Tuning and analysis utilities. Technical report, LALP-99–205, Los Alamos National Laboratory Publication, 1999.

[10] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka. Physis: an implicitly parallel programming model for stencil computations on large-scale gpu-accelerated supercomputers. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–12. IEEE, 2011.

[11] C. Matthias, S. Olaf, and B. Helmar. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 676–687. IEEE, 2011.

[12] P. Micikevicius. 3d finite difference computation on gpus using cuda. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 79–84. ACM, 2009.

[13] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The pochoir stencil compiler. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 117–128. ACM, 2011.

[14] D. Unat, X. Cai, and S. B. Baden. Mint: realizing cuda performance in 3d stencil methods with annotated c. In *Proceedings of the international conference on Supercomputing*, pages 214–224. ACM, 2011.

[15] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 1–27. IEEE Computer Society, 1998.